# CS 152 Computer Architecture and Engineering

# Lecture 5 - Pipelining II

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

**http://www.eecs.berkeley.edu/~krste**
**http://inst.eecs.berkeley.edu/~cs152**

# Last time in Lecture 4

- Pipelining increases clock frequency, while growing CPI more slowly, hence giving greater performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Increases because of pipeline bubbles

Reduces because fewer logic gates on critical paths between flip-flops

- Pipelining of instructions is complicated by HAZARDS:
  - Structural hazards (two instructions want same hardware resource)
  - Data hazards (earlier instruction produces value needed by later instruction)
  - Control hazards (instruction changes control flow, e.g., branches or exceptions)
- Techniques to handle hazards:
  - Interlock (hold newer instruction until older instructions drain out of pipeline and write back results)
  - Bypass (transfer value from older instruction to newer instruction as soon as available somewhere in machine)
  - Speculate (guess effect of earlier instruction)

# Control Hazards

- What do we need to calculate next PC?
  - For Jumps
    - » Opcode, offset and PC
  - For Jump Register
    - » Opcode and Register value
  - For Conditional Branches
    - » Opcode, PC, Register (for condition), and offset
  - For all other instructions
    - » Opcode and PC
      - have to know it’s not one of above

# PC Calculation Bubbles

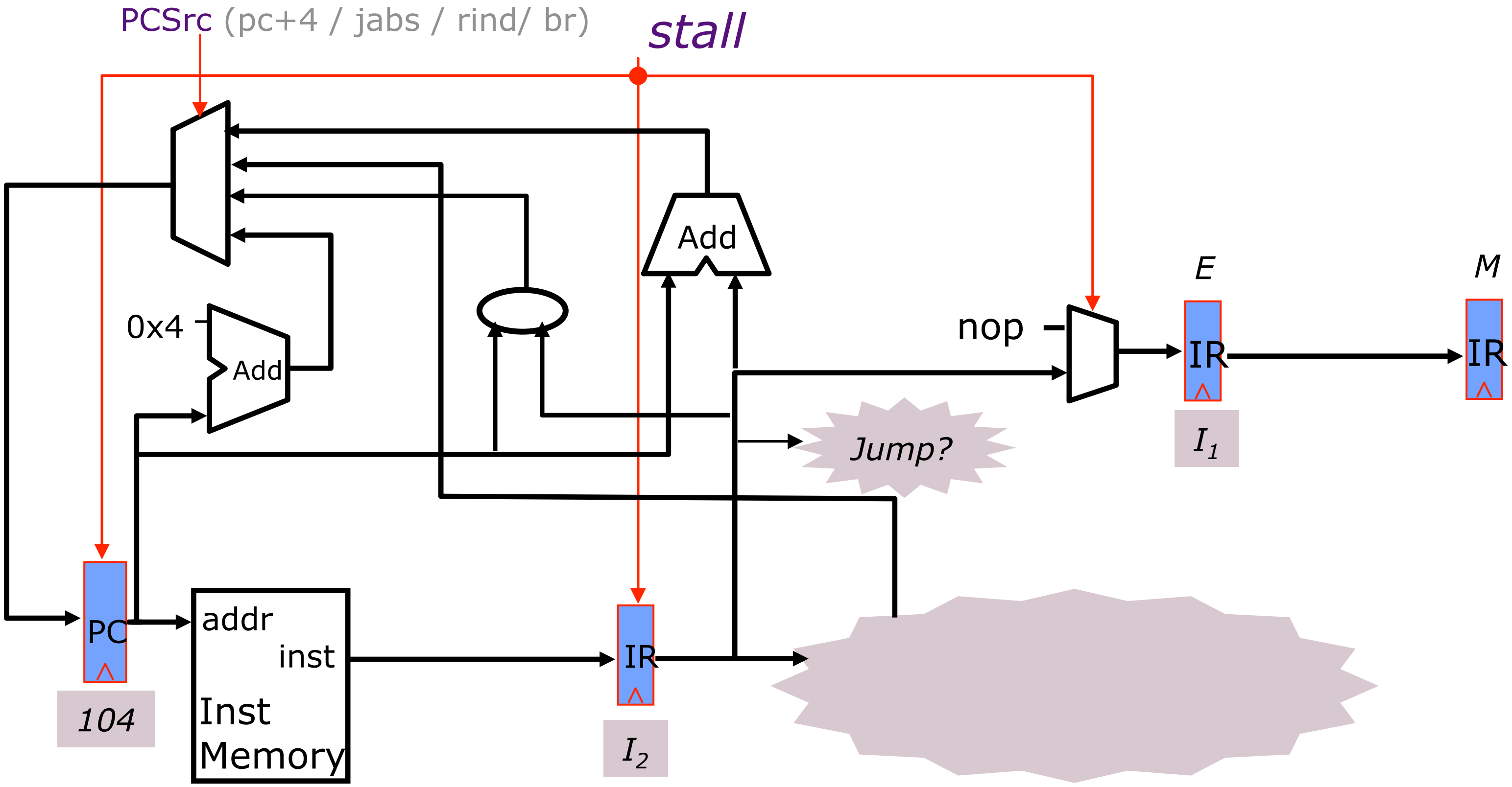
*(assuming no branch delay slots for now)*

| | *time* | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . | | |
| $(I_1)$ r1 ← (r0) + 10 | $IF_1$ | $ID_1$ | $EX_1$ | $MA_1$ | $WB_1$ | | | | | | |
| $(I_2)$ r3 ← (r2) + 17 | | $IF_2$ | $IF_2$ | $ID_2$ | $EX_2$ | $MA_2$ | $WB_2$ | | | | |
| $(I_3)$ | | | | $IF_3$ | $IF_3$ | $ID_3$ | $EX_3$ | $MA_3$ | $WB_3$ | | |
| $(I_4)$ | | | | | | $IF_4$ | $IF_4$ | $ID_4$ | $EX_4$ | $MA_4$ | $WB_4$ |

| | | *time* | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . | | |
| | IF | $I_1$ | nop | $I_2$ | nop | $I_3$ | nop | $I_4$ | | | | |
| *Resource Usage* | ID | | $I_1$ | nop | $I_2$ | nop | $I_3$ | nop | $I_4$ | | | |
| | EX | | | $I_1$ | nop | $I_2$ | nop | $I_3$ | nop | $I_4$ | | |
| | MA | | | | $I_1$ | nop | $I_2$ | nop | $I_3$ | nop | $I_4$ | |
| | WB | | | | | $I_1$ | nop | $I_2$ | nop | $I_3$ | nop | $I_4$ |

*nop* ⇒ *pipeline bubble*

# Speculate next address is PC+4



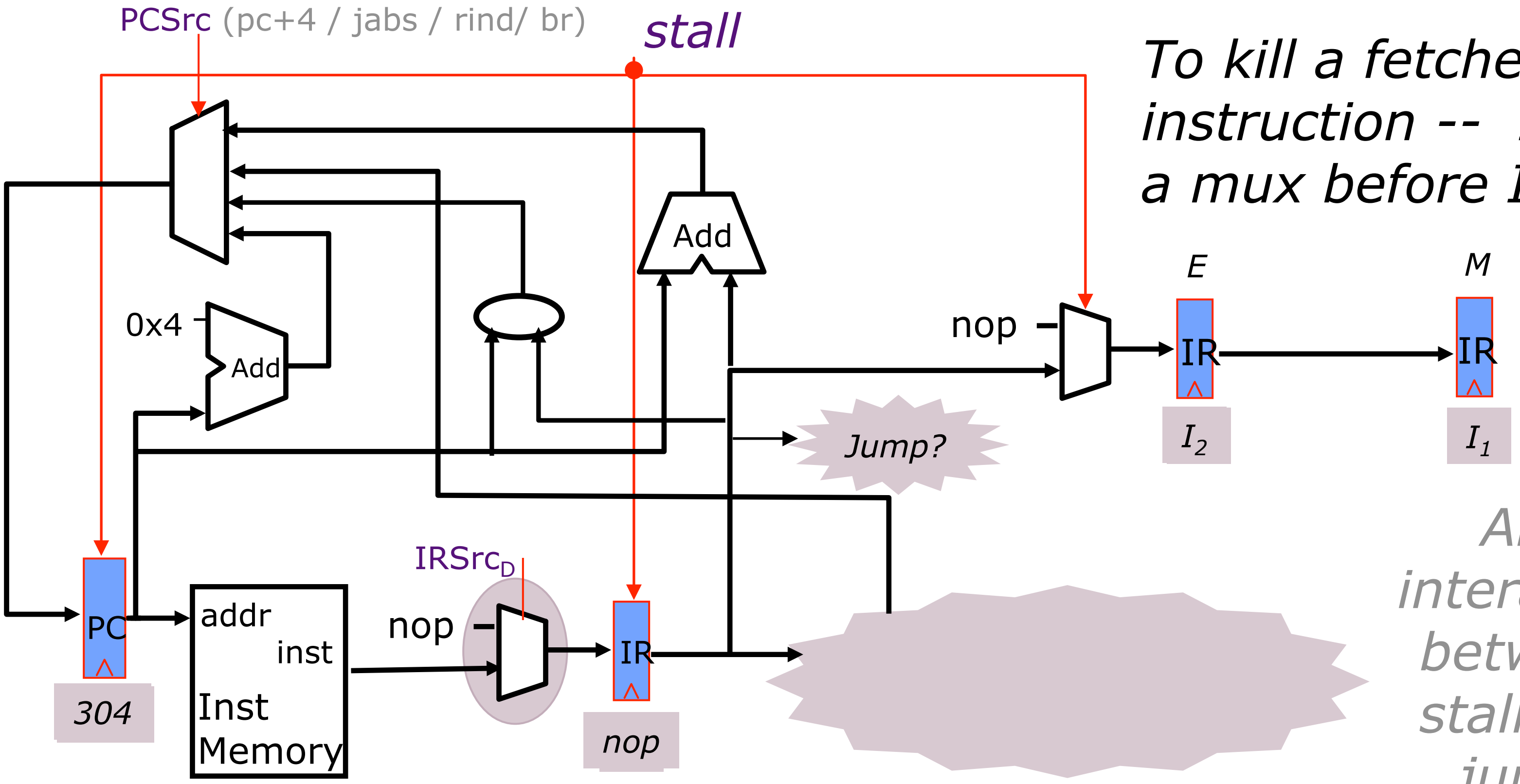
| | | |
|---|---|---|
| $I_1$ | 096 | ADD |
| $I_2$ | 100 | J 304 |
| $I_3$ | ~~104~~ | ~~ADD~~ |
| $I_4$ | 304 | ADD |

*kill*

A jump instruction kills (not stalls) the following instruction

*How?*

# Pipelining Jumps



*To kill a fetched instruction -- Insert a mux before IR*

*Any interaction between stall and jump?*

| | | |
|---|---|---|
| $I_1$ | 096 | ADD |
| $I_2$ | 100 | J 304 |
| $I_3$ | ~~104~~ | ~~ADD~~ |
| $I_4$ | 304 | ADD |

*kill*

$IRSrc_D$ = *Case* $opcode_D$
J, JAL ⇒ nop
... ⇒ IM

# Jump Pipeline Diagrams

*time*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| ($I_1$) 096: ADD | $IF_1$ | $ID_1$ | $EX_1$ | $MA_1$ | $WB_1$ | | | | |
| ($I_2$) 100: J 304 | | $IF_2$ | $ID_2$ | $EX_2$ | $MA_2$ | $WB_2$ | | | |
| ($I_3$) 104: ADD | | | $IF_3$ | nop | nop | nop | nop | | |
| ($I_4$) 304: ADD | | | | $IF_4$ | $ID_4$ | $EX_4$ | $MA_4$ | $WB_4$ | |

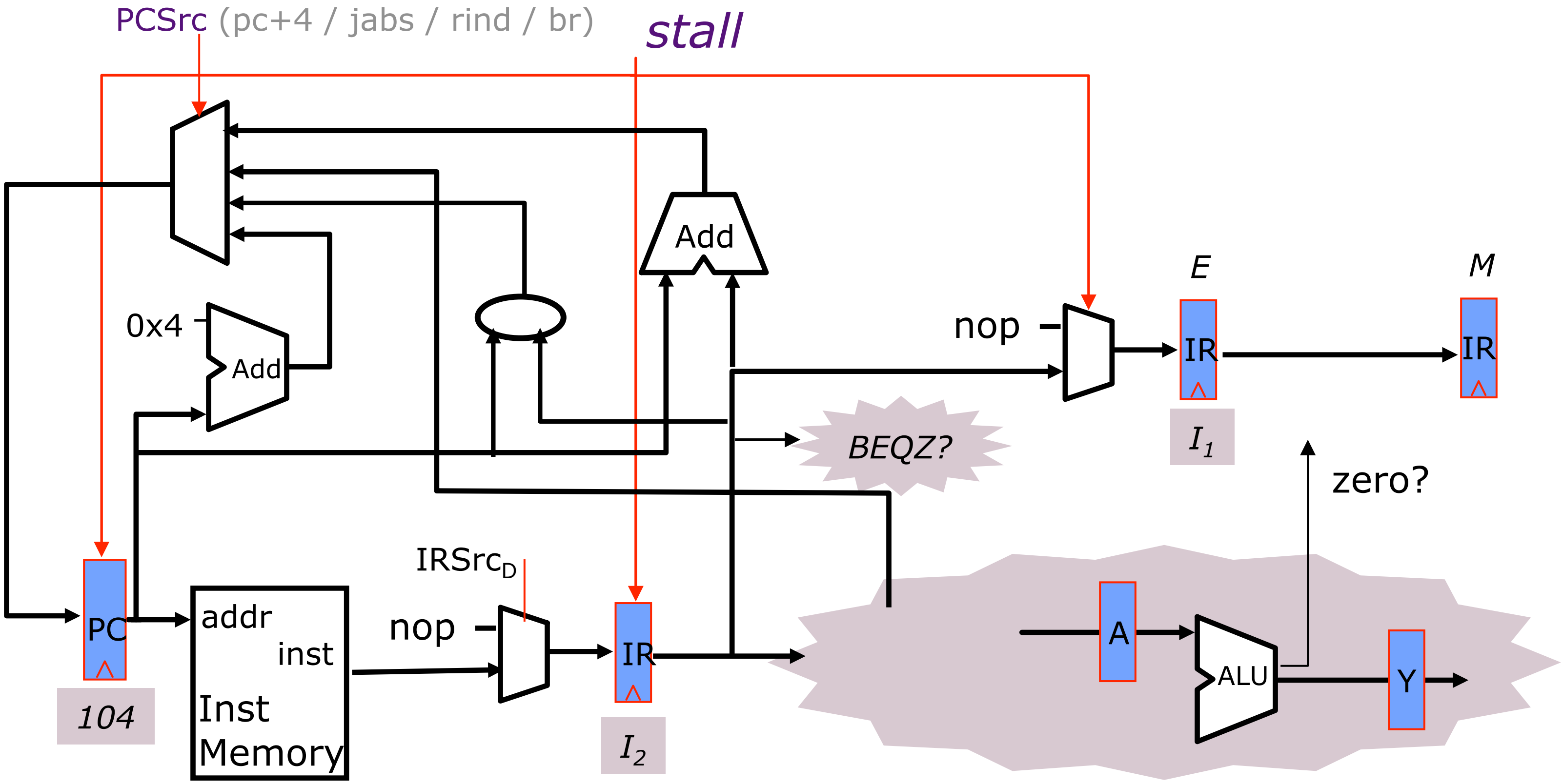
*Resource Usage*

*time*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| IF | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | | | |
| ID | | $I_1$ | $I_2$ | nop | $I_4$ | $I_5$ | | | |
| EX | | | $I_1$ | $I_2$ | nop | $I_4$ | $I_5$ | | |
| MA | | | | $I_1$ | $I_2$ | nop | $I_4$ | $I_5$ | |
| WB | | | | | $I_1$ | $I_2$ | nop | $I_4$ | $I_5$ |

*nop* ⇒ *pipeline bubble*

# Pipelining Conditional Branches

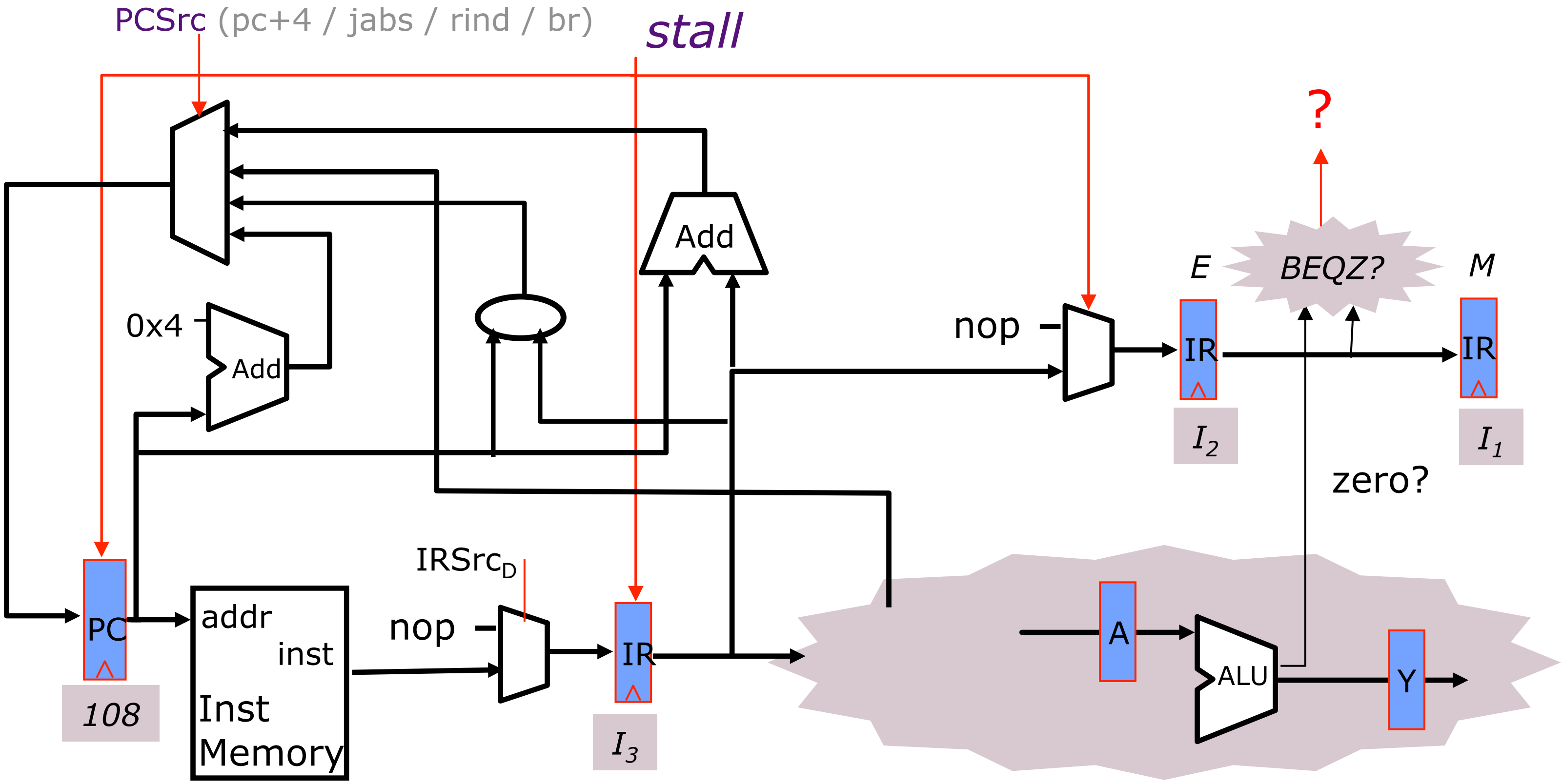


| | | |
|---|---|---|
| $I_1$ | 096 | ADD |
| $I_2$ | 100 | BEQZ r1 +200 |
| $I_3$ | 104 | ADD |
| $I_4$ | 304 | ADD |

Branch condition is not known until the execute stage

*what action should be taken in the decode stage ?*

# Pipelining Conditional Branches



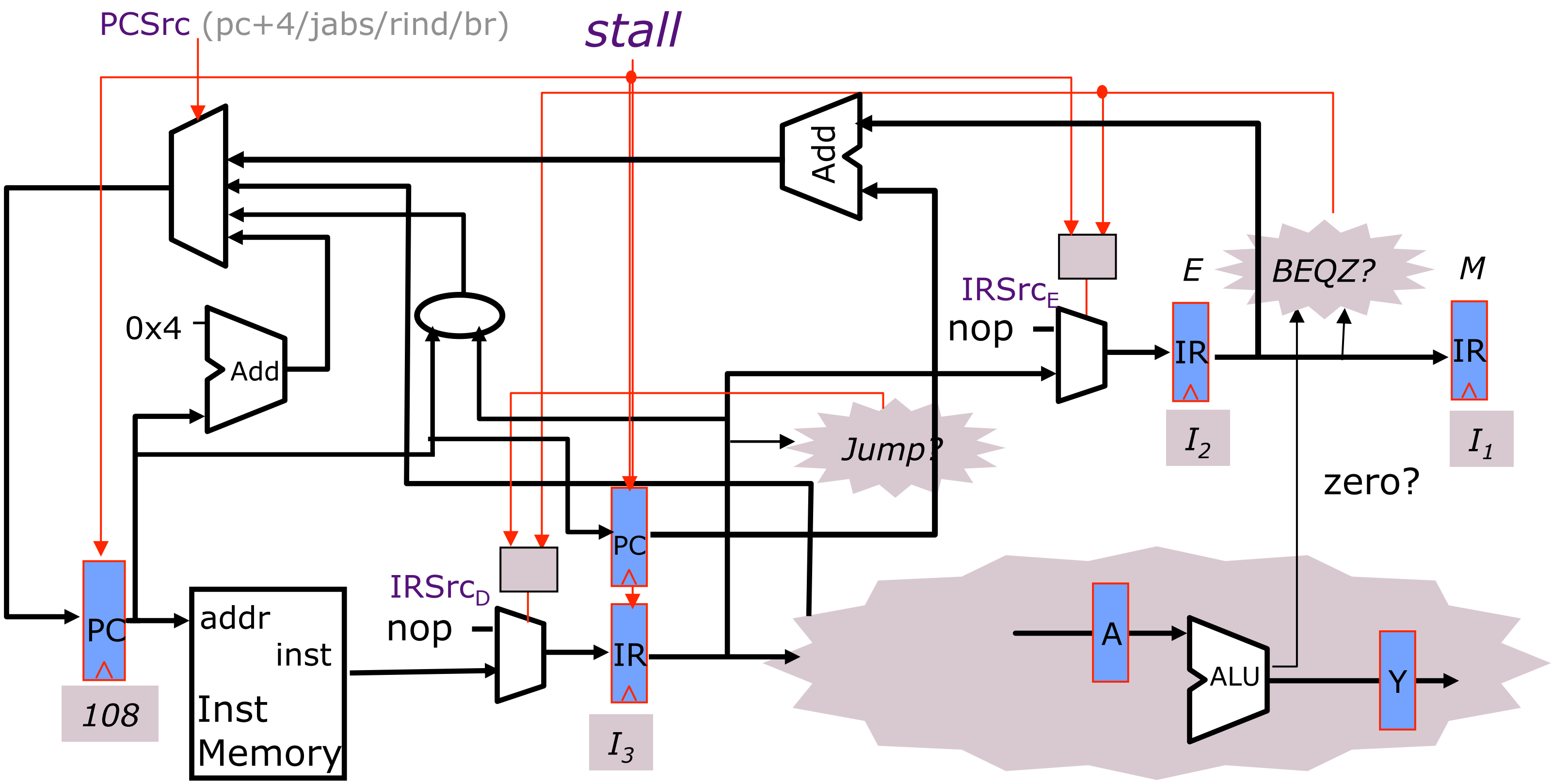
| | | |
|---|---|---|
| $I_1$ | 096 | ADD |
| $I_2$ | 100 | BEQZ r1 +200 |
| $I_3$ | 104 | ADD |
| $I_4$ | 304 | ADD |

If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ *stall signal is not valid*

# Pipelining Conditional Branches



| | | |
|---|---|---|
| $I_1$ | 096 | ADD |
| $I_2$ | 100 | BEQZ r1 +200 |
| $I_3$ | 104 | ADD |
| $I_4$ | 304 | ADD |

If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ *stall signal is not valid*

# New Stall Signal

$$\begin{aligned}\text{stall} = ( \quad & ((rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D \\ & + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D \\ & ) \;.\; !((opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)\end{aligned}$$

Don’t stall if the branch is taken. Why?

Instruction at the decode stage is invalid

# Control Equations for PC and IR Muxes

```
PCSrc = Case $opcode_E$
        BEQZ.z, BNEZ.!z     ⇒ br
        ...                 ⇒
                    Case $opcode_D$
                        J, JAL       ⇒  jabs
                        JR, JALR     ⇒  rind
                        ...          ⇒  pc+4
```

```
$IRSrc_D$ = Case $opcode_E$
        BEQZ.z, BNEZ.!z     ⇒ nop
        ...                 ⇒
                    Case $opcode_D$
                        J, JAL, JR, JALR ⇒ nop
                        ...              ⇒ IM
```

```
$IRSrc_E$ = Case $opcode_E$
        BEQZ.z, BNEZ.!z     ⇒ nop
        ...                 ⇒ stall.nop + !stall.$IR_D$
```

*Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction*

# Branch Pipeline Diagrams
## (resolved in execute stage)

| | *time* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
| ($I_1$) 096: ADD | IF$_1$ | ID$_1$ | EX$_1$ | MA$_1$ | WB$_1$ | | | | |
| ($I_2$) 100: BEQZ +200 | | IF$_2$ | ID$_2$ | EX$_2$ | MA$_2$ | WB$_2$ | | | |
| ($I_3$) 104: ADD | | | IF$_3$ | ID$_3$ | nop | nop | nop | | |
| ($I_4$) 108: | | | | IF$_4$ | nop | nop | nop | nop | |
| ($I_5$) 304: ADD | | | | | IF$_5$ | ID$_5$ | EX$_5$ | MA$_5$ | WB$_5$ |

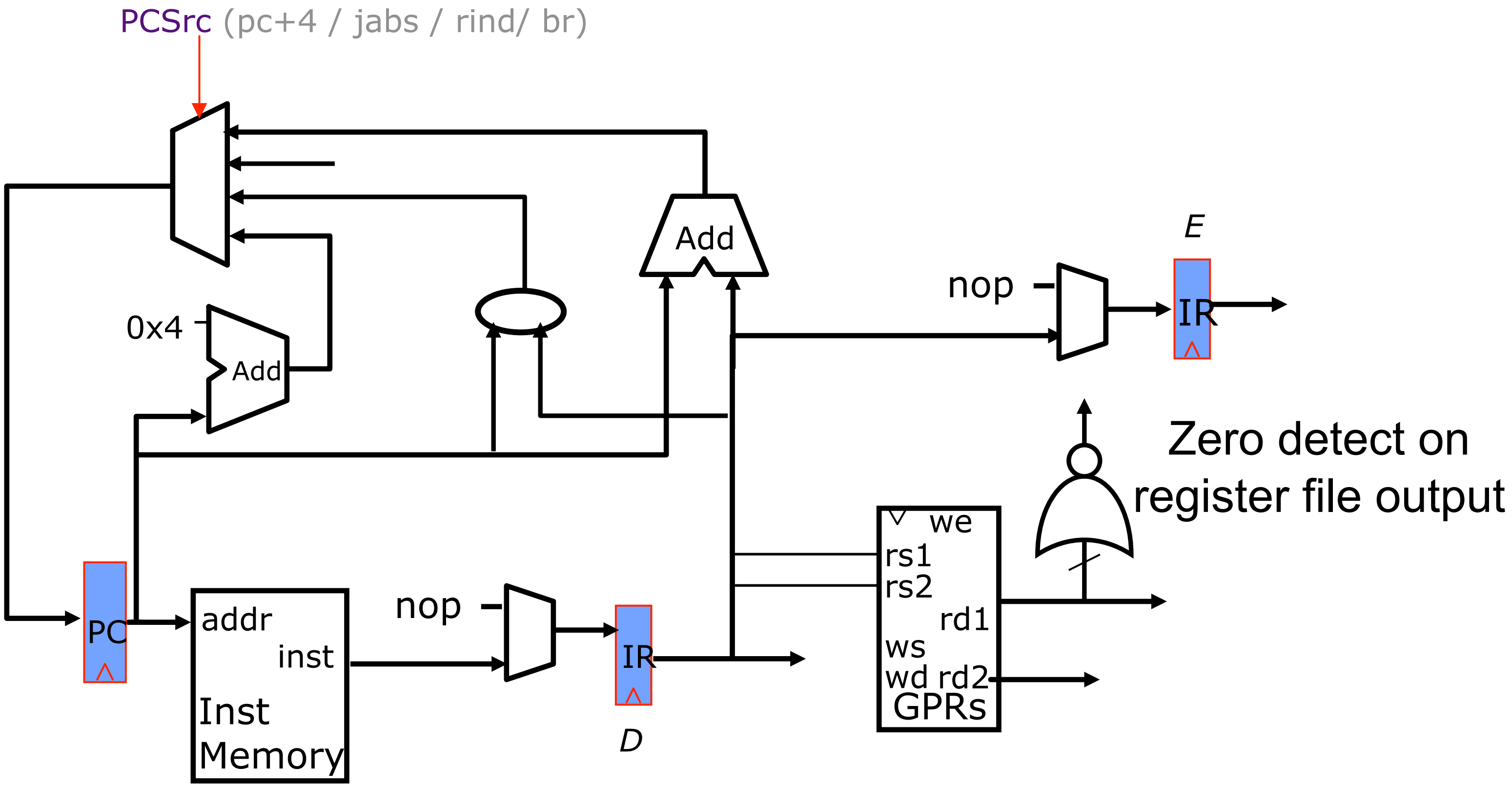
*Resource Usage*

| | *time* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
| IF | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | | | |
| ID | | $I_1$ | $I_2$ | $I_3$ | nop | $I_5$ | | | |
| EX | | | $I_1$ | $I_2$ | nop | nop | $I_5$ | | |
| MA | | | | $I_1$ | $I_2$ | nop | nop | $I_5$ | |
| WB | | | | | $I_1$ | $I_2$ | nop | nop | $I_5$ |

*nop* ⇒ *pipeline bubble*

# Reducing Branch Penalty
## (resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage


PCSrc (pc+4 / jabs / rind/ br)
Add
0x4
Add
nop
E
IR
Zero detect on register file output
we
rs1
rs2
rd1
ws
wd rd2
GPRs
PC
addr
inst
Inst Memory
nop
IR
D

*Pipeline diagram now same as for jumps*

# Branch Delay Slots (expose control hazard to software)

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

| | | |
|---|---|---|
| $I_1$ | 096 | ADD |
| $I_2$ | 100 | BEQZ r1 +200 |
| $I_3$ | 104 | ADD ← |
| $I_4$ | 304 | ADD |

*Delay slot instruction executed regardless of branch outcome*

- Other techniques include more advanced branch prediction, which can dramatically reduce the branch penalty... *to come later*

# Branch Pipeline Diagrams
## (branch delay slot)

*time*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| ($I_1$) 096: ADD | $IF_1$ | $ID_1$ | $EX_1$ | $MA_1$ | $WB_1$ | | | | |
| ($I_2$) 100: BEQZ +200 | | $IF_2$ | $ID_2$ | $EX_2$ | $MA_2$ | $WB_2$ | | | |
| ($I_3$) 104: ADD | | | $IF_3$ | $ID_3$ | $EX_3$ | $MA_3$ | $WB_3$ | | |
| ($I_4$) 304: ADD | | | | $IF_4$ | $ID_4$ | $EX_4$ | $MA_4$ | $WB_4$ | |

*time*

*Resource Usage*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| IF | $I_1$ | $I_2$ | $I_3$ | $I_4$ | | | | | |
| ID | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | | | | |
| EX | | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | | | |
| MA | | | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | | |
| WB | | | | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | |

# Why an Instruction may not be dispatched every cycle (CPI>1)

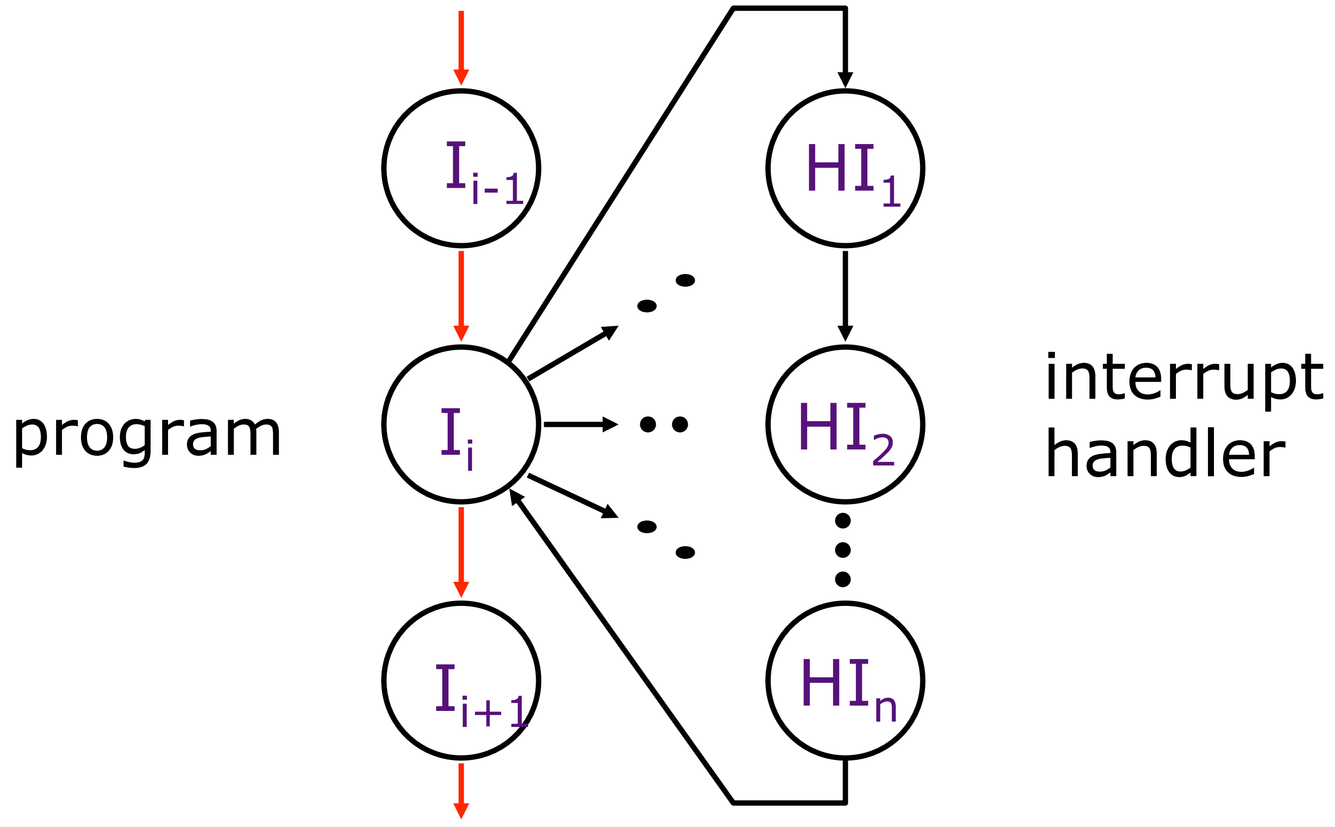
- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II (pipeline interlocks added in hardware)
    - » MIPS:"**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages"
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

*Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler. NOPs not counted in useful CPI (alternatively, increase instructions/program)*

# CS152 Administrivia

- PS1/Lab1 due start of class Thursday Feb 11
- Quiz 1, Tuesday Feb 16

# Interrupts:
## altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program’s point of view.

# Causes of Interrupts

Interrupt: an *event* that requests the attention of the processor

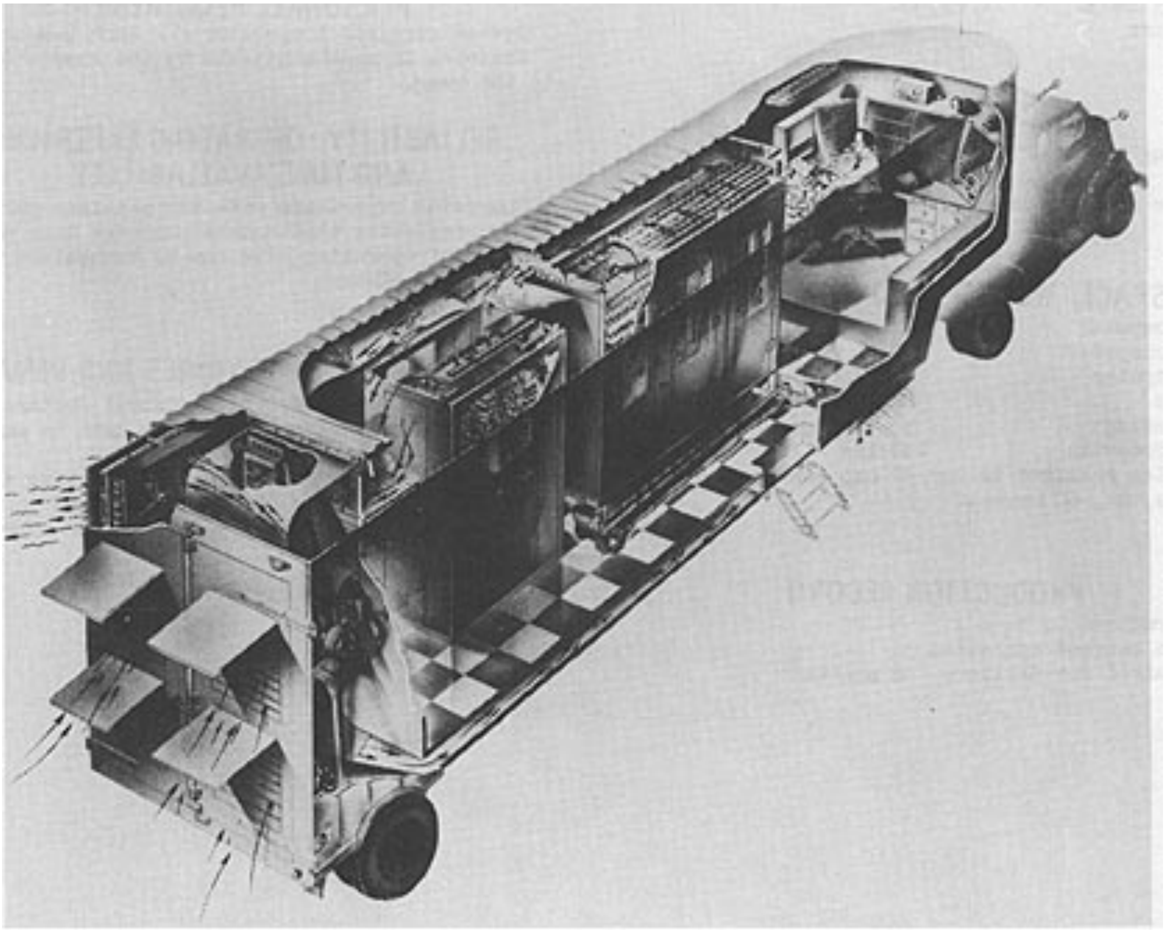
- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a. exceptions)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults, TLB misses, protection violations
  - *traps:* system calls, e.g., jumps into kernel

# History of Exception Handling

- First system with exceptions was Univac-I, 1951
  - Arithmetic overflow would either
    - » 1. trigger the execution a two-instruction fix-up routine at address 0, or
    - » 2. at the programmer's option, cause the computer to stop
  - Later Univac 1103, 1955, modified to add external interrupts
    - » Used to gather real-time wind tunnel data
- First system with I/O interrupts was DYSEAC, 1954
  - Had two program counters, and I/O signal caused switch between two PCs
  - Also, first system with DMA (direct memory access by I/O device)

*[Courtesy Mark Smotherman]*

# DYSEAC, first mobile computer!



- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

*[Courtesy Mark Smotherman]*

# Asynchronous Interrupts:

## invoking the interrupt handler

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

- When the processor decides to process the interrupt
  - It stops the current program at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (*precise interrupt)*
  - It saves the PC of instruction $I_i$ in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

# Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

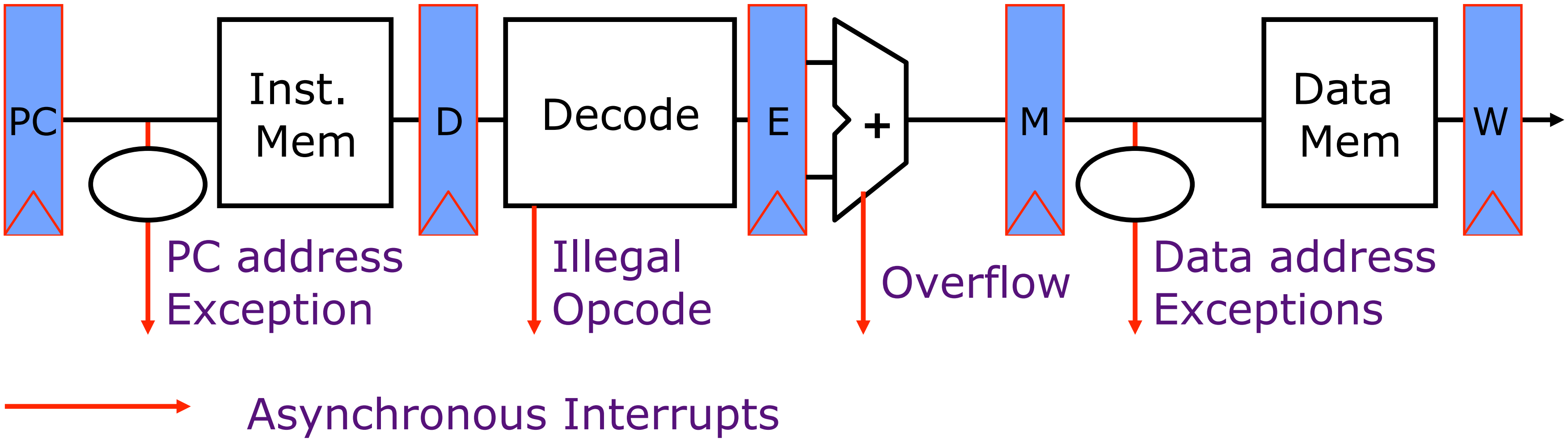
# Synchronous Interrupts

- A synchronous interrupt (exception) is caused by a *particular instruction*

- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions

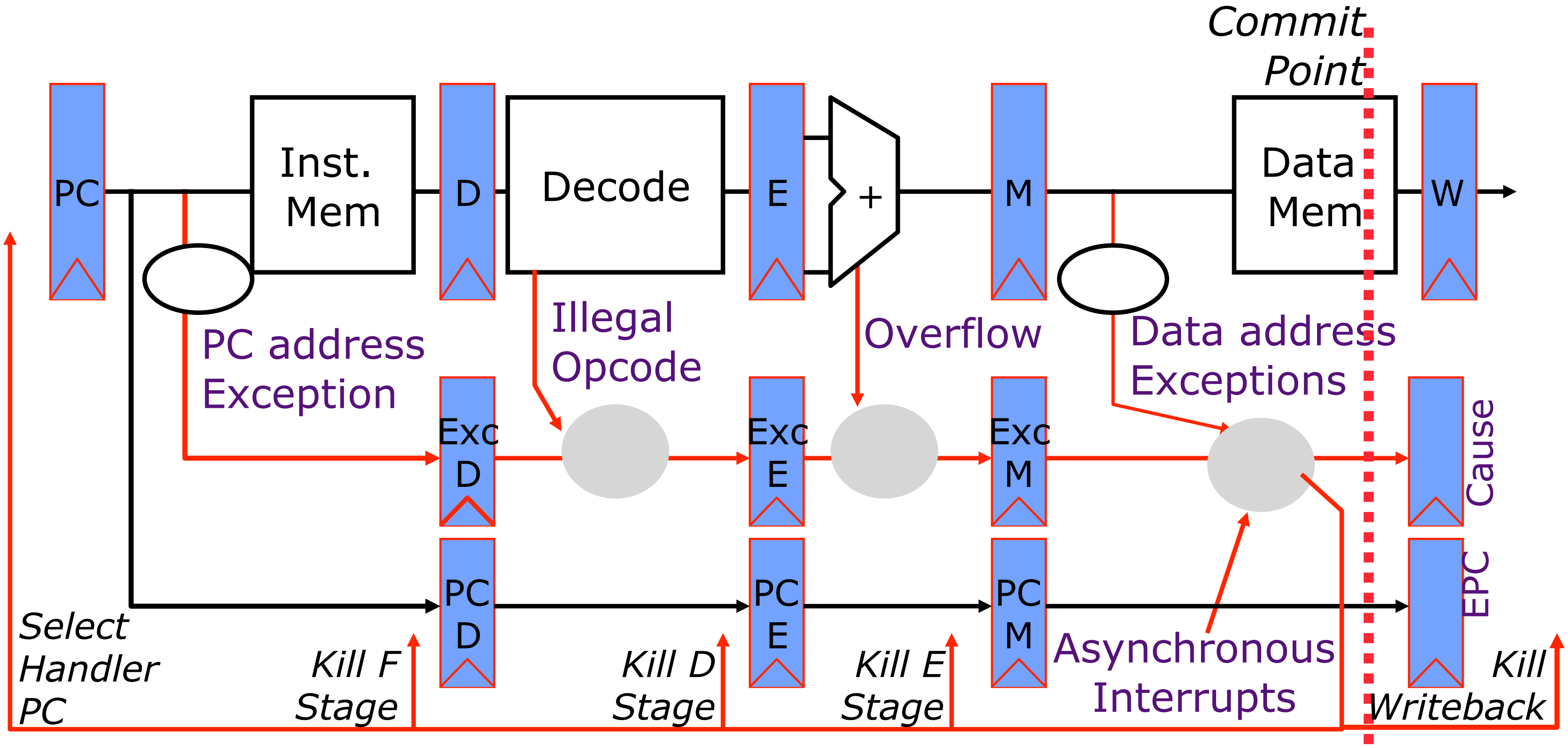
- In the case of a system call trap, the instruction is considered to have been completed
  - a special jump instruction involving a change to privileged kernel mode

# Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

# Exception Handling 5-Stage Pipeline



# Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)

- Exceptions in earlier pipe stages override later exceptions *for a given instruction*

- Inject external interrupts at commit point (override others)

- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

# Speculating on Exceptions

- Prediction mechanism
  - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline

- Bypassing allows use of uncommitted instruction results by following instructions

# Exception Pipeline Diagram

*time*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| ($I_1$) 096: ADD | $IF_1$ | $ID_1$ | $EX_1$ | $MA_1$ | nop | | *overflow!* | | |
| ($I_2$) 100: XOR | | $IF_2$ | $ID_2$ | $EX_2$ | nop | nop | | | |
| ($I_3$) 104: SUB | | | $IF_3$ | $ID_3$ | nop | nop | nop | | |
| ($I_4$) 108: ADD | | | | $IF_4$ | nop | nop | nop | nop | |
| ($I_5$) Exc. Handler code | | | | | $IF_5$ | $ID_5$ | $EX_5$ | $MA_5$ | $WB_5$ |

*time*

| | | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|---|
| | IF | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | | | |
| *Resource Usage* | ID | | $I_1$ | $I_2$ | $I_3$ | nop | $I_5$ | | | |
| | EX | | | $I_1$ | $I_2$ | nop | nop | $I_5$ | | |
| | MA | | | | $I_1$ | nop | nop | nop | $I_5$ | |
| | WB | | | | | nop | nop | nop | nop | $I_5$ |

# Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252